Petri Nets and Linear Logic: a case study for logic programming

Iliano Cervesato*
Department of Computer Science – Carnegie Mellon University
5000 Forbes Avenue – Pittsburgh, PA, 15213-3891
Phone: (412)-268-1413 – Fax: (412)-268-5576
iliano@cs.cmu.edu

Abstract

The paper reports on an experiment with the major linear logic programming languages defined in the recent years: Lolli, LO and Forum. As a case study, we consider the representation of a class of Petri nets, P/T^{ω} nets, in each of these languages.

Keywords: Petri nets, multiset rewriting systems, linear logic, logic programming.

1 Introduction

The relationship between linear logic [4] and Petri Nets [8] has been a major object of investigation [2, 3, 6]. It was soon noticed that the monoidal structure of multisets can serve as an abstract link between Petri nets (where markings are multisets and transition are rewrite rules on multisets) and the multiplicative fragment of linear logic (where either $\otimes$ or $\parr$ is the operation of the monoid and $\mathbf{1}$ or $\bot$ is its identity). See [3] for a precise account of this relationship.

On the other hand, a number of fragments of linear logic have been cast into logic programming languages with the aim of capturing some of the expressive power provided by this new logic. We consider three such proposals, Lolli [5], LO [1] and Forum [7], and compare their flexibility precisely on the encoding of Petri nets (or equivalently of multiset rewriting systems). Considerations about the amenability of these languages to efficient implementations are outside the scope of this paper.

*This research was completed during an extended visit of the author at the *Department of Computer Science* of *Carnegie Mellon University*. His permanent address is: *Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy*.

Appeared in the *Proceedings of the 1995 Joint Conference on Declarative Programming — GULP–PRODE'95* (M. Alpuente, M.I. Sessa editors), pp. 313–318, Palladio Press, Marina di Vietri, Italy, 11–14 September 1995.

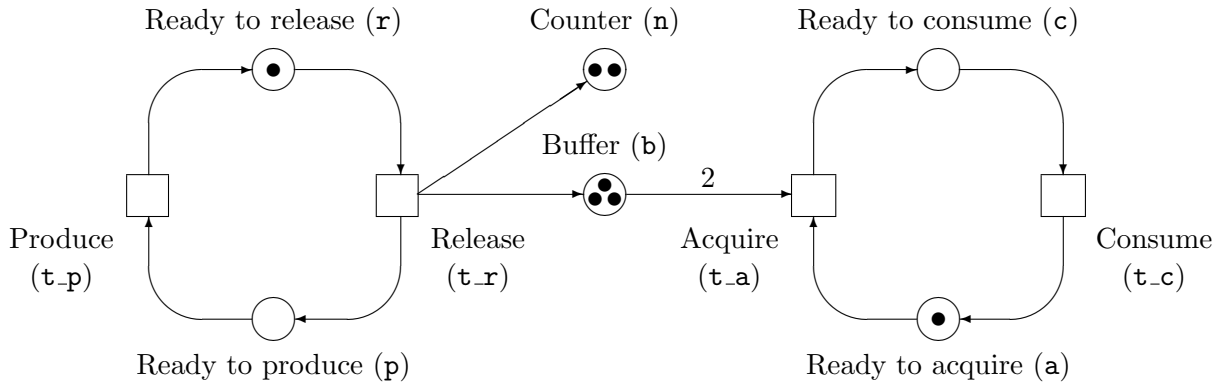


Figure 1: A producer/consumer net

2 P/T$^\omega$ nets

A *multiset rewriting system* (*MRS*) $\mathcal{M}$ over a set S is a set of *multiset rewrite rules* $t = {\bullet t} \triangleright t{\bullet}$ with ${\bullet t}$ and $t{\bullet}$ multisets over S called respectively the *preset* and the *postset* of t. Given a multiset M, a rule t in $\mathcal{M}$ is *enabled* at M if ${\bullet t}$ is a submultiset of M. If t is enabled at M, the *application* of t on M produces the multiset $M' = M - {\bullet t} + t{\bullet}$, $+$ and $-$ being the multiset union and difference respectively. We write in this case $M \rhd^t_{\mathcal{M}} M'$ (or $M \rhd_{\mathcal{M}} M'$). We use $\rhd^*_{\mathcal{M}}$ for the reflexive and transitive closure of $\rhd_{\mathcal{M}}$.

A *place/transition net with infinite place capacities*, *P/T$^\omega$ net* for short, is a pair $N = (S, \mathcal{M})$ where S is a set and $\mathcal{M}$ is an MRS over S. We require either S or $\mathcal{M}$ to be non-empty. The elements of S are called *places* and the elements of $\mathcal{M}$ are called *transitions*. A *P/T$^\omega$ system* is a pair $\mathcal{N} = (N, M)$ where $N = (S, \mathcal{M})$ is a P/T$^\omega$ net and M is a multiset over S called a *marking*.

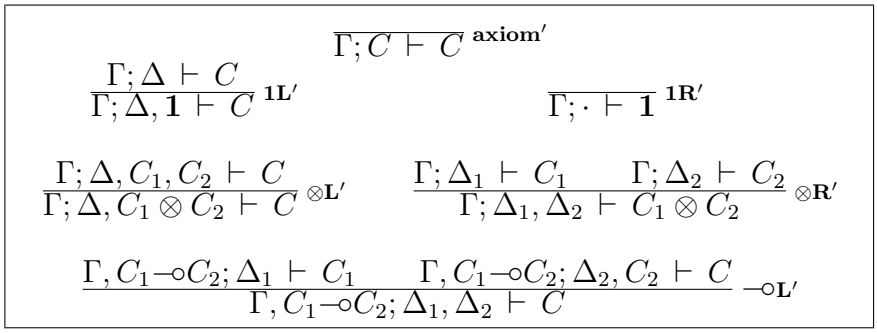
Figure 1 describes a producer-consumer system by means of the usual graphical representation for Petri nets. The left cycle represents the producer, that releases one item per cycle, puts it in the common buffer and increments by one the counter. The consumer (on the right) extracts two items at a time from the buffer and consumes them. We will use this example as a benchmark for our implementations. In section 4, we will rely on the abbreviations shown in parentheses.

The dynamic behavior of a P/T$^\omega$ net is inherited from the underlying MRS. In this context, the application of an enabled transition (i.e. a multiset rewrite rule) to a marking is referred to as the *firing* of that transition.

3 Encoding Petri nets in linear logic

We describe in this section two encodings of P/T$^\omega$ systems in multiplicative linear logic. The first uses the language $\mathcal{L}^\otimes$ to encode multisets by means of the connectives $\otimes$ and $\mathbf{1}$. Dually, $\mathcal{L}^{\parr}$ relies on $\parr$ and $\bot$.

A sequent for $\mathcal{L}^\otimes$ has the form $\Gamma; \Delta \vdash C$ where C is a possibly empty conjunction of atomic formulas (C-formula), Δ is a multiset of C-formulas and Γ is a set of linear implications with C-formulas as antecendent and consequent (I-formula). The rules



$$\frac{}{\Gamma; C \vdash C}\ \mathbf{axiom'}$$

$$\frac{\Gamma; \Delta \vdash C}{\Gamma; \Delta, \mathbf{1} \vdash C}\ \mathbf{1L'} \qquad \frac{}{\Gamma; \cdot \vdash \mathbf{1}}\ \mathbf{1R'}$$

$$\frac{\Gamma; \Delta, C_1, C_2 \vdash C}{\Gamma; \Delta, C_1 \otimes C_2 \vdash C}\ \otimes\mathbf{L'} \qquad \frac{\Gamma; \Delta_1 \vdash C_1 \qquad \Gamma; \Delta_2 \vdash C_2}{\Gamma; \Delta_1, \Delta_2 \vdash C_1 \otimes C_2}\ \otimes\mathbf{R'}$$

$$\frac{\Gamma, C_1 \multimap C_2; \Delta_1 \vdash C_1 \qquad \Gamma, C_1 \multimap C_2; \Delta_2, C_2 \vdash C}{\Gamma, C_1 \multimap C_2; \Delta_1, \Delta_2 \vdash C}\ \multimap\mathbf{L'}$$

Figure 2: The system $\mathrm{MILL}^{P/T}$ for $\mathcal{L}^{\otimes}$.

for the resulting system are presented in figure 3. $\mathrm{MILL}^{P/T}$ is proved in [3] to be equivalent to the usual system for full linear logic when restricted to formulas in $\mathcal{L}^{\otimes}$.

In order to represent $\mathrm{P/T}^{\omega}$ nets in $\mathcal{L}^{\otimes}$, we associate to every element in the universe of a multiset a propositional letter. A multiset is then mapped to a C-formula consisting of the multiplicative conjunction of its members. A rewrite rule is represented by an I-formula built upon the representation of its preset and postset. We will call this representation the *conjunctive encoding* and denote it with $\ulcorner \cdot \urcorner$, being $\llcorner \cdot \lrcorner$ the inverse function. We have the following result [3].

Theorem 1 (*Soundness and completeness of the conjunctive encoding of MRSs*)
Let M and M' be two multisets and $\mathcal{M}$ a multiset rewriting system over a common universe. Then, if $M \rhd^*_{\mathcal{M}} M'$, there exist a derivation of $\ulcorner \mathcal{M} \urcorner; \ulcorner M \urcorner \vdash \ulcorner M' \urcorner$.

Let Γ, Δ and C be an I-context, a C-context and a C-formula respectively, Then, if $\mathcal{PT}$ is a derivation of $\Gamma; \Delta \vdash C$, then $\llcorner \Delta \lrcorner \rhd^*_{\llcorner \Gamma \lrcorner} \llcorner C \lrcorner$. ∎

The dual representation in $\mathcal{L}^{\parr}$ consists in mapping multisets to D-formulas, i.e. the set of formulas freely generated from $\bot$ and $\parr$. We call this representation the *disjunctive encoding* $\mathrm{P/T}^{\omega}$ systems [3].

4 Implementations in linear logic programming

We will now consider in turn three logic programming languages based on linear logic and show how the notions presented in the previous section can be effectively implemented by using each of them. The languages we chose are Lolli [5], LO [1] and Forum [7]. These three languages differ for the fragment of linear logic they are based upon. Consequently, the encoding of MRSs, and therefore of $\mathrm{P/T}^{\omega}$ systems, will be different in each language. We foresay that the implementation will become more direct and elegant as we move from Lolli to LO and finally to Forum. As a benchmark, we use the producer-consumer example of figure 1.

4.1 Implementation in Lolli

The fragment of linear logic that Lolli [5] makes available as a programming language is the first-order language freely generated from $\top$, $\&$, $\multimap$, $\Rightarrow$, where $A \Rightarrow B$ is

Generic rules

```
rewrite L K :- {load L K}.

load (X::L) K :- (item X -o load L K).
load nil    K :- rew K.

rew K :- unload K.

unload (X::L) :- item X, unload L.
unload nil.
```

Specific rules

```
rew K :- item p, (item r -o rew K).                          % t_p
rew K :- item r, (item p -o (item b -o (item n -o rew K))). % t_r
rew K :- item b, item b, item a, (item c -o rew K).          % t_a
rew K :- item c, (item a -o rew K).                          % t_c
```

Example query

```
?- rewrite r::n::n::b::b::b::a::nil  p::n::n::n::b::c::nil.
```

Figure 3: A Lolli encoding of the producer/consumer net

defined to be equal to $!A \multimap B$, and $\forall^1$. Lolli formalizes intuitionistic provability for this fragment of linear logic over sequents of the form $\Gamma; \Delta \vdash G$, where Γ and Δ are two multisets of formulas called the *unbound* and the *bound context* respectively and G is the *goal formula*.

For our purpose, we will rely exclusively on the connectives $\otimes$, $\multimap$ and $!$ (written respectively `,`, `-o` or `:-`, and $\{\ldots\}$ in the concrete syntax). Operationally, $\otimes$ requires the bound context to be split among its two subgoals, $D \multimap G$ involves adding D to the bound context in order to prove G, and $!G$ succeeds if G is provable in an empty bound context. When a clause or a fact from the bound context is used, it is deleted.

Lolli does not provide the tools for a direct implementation of neither the conjunctive nor the disjunctive encoding. Therefore, an elegant solution cannot be achieved within Lolli. However, we can simulate the former by representing a generic rule $t = \{\!\{a_1, \ldots, a_n\}\!\} \triangleright \{\!\{b_1, \ldots, b_m\}\!\}$ as the clause `rew K :- a1, ..., an, (b1 -o (b2 -o ... (bm -o rew K)...))`. When called, the first part of this clause (`a1, ..., an`) consumes the preset of t while the remaining part asserts its postset and finally calls `rew K` recursively for the application of another rule.

The full program, adapted from [5], is shown in figure 3. Markings are given two representations: the users views them externally as lists and the program represents internally their constituents as individual facts in the bound context. A multiset rewriting sequence is performed in three stages: first the list representing the initial marking is downloaded into the bound context (`load`), then the rewrite rules are applied (`rew`), and finally the resulting multiset is encoded back into a list (`unload`).

[1]The syntax of Lolli can be enriched by permitting the following connectives in a goal position: $\oplus$, $\mathbf{1}$, $\otimes$, $!$ and $\exists$. The resulting formulas are called (linear) *hereditary Harrop formulas*.

```
                                                          Specific rules
r          :- p.                          % t_p
p @ b @ c  :- r.                          % t_r
c          :- b @ b @ a.                  % t_a
a          :- c.                          % t_c
r @ n @ n @ b @ b @ b @ a          @ bot @ stop.
                                                          Example query
?- p @ n @ n @ n @ b @ c           @ bot @ stop.
```

Figure 4: An LO encoding of the producer/consumer net

4.2 Implementation in LO

LO mechanizes the fragment of linear logic specified by the following grammar rules:

$$V \;::=\; A \;\mid\; V_1 \parr V_2 \qquad \text{(Head formulas)}$$
$$G \;::=\; A \;\mid\; \top \;\mid\; G_1 \,\&\, G_2 \;\mid\; G_1 \parr G_2 \qquad \text{(Goal formulas)}$$
$$D \;::=\; V \;\mid\; G \multimap V \qquad \text{(Clauses)}$$

where A ranges over atomic formulas (for the purpose of our example, it will be sufficient to deal with the propositional fragment of LO).

The operational semantics of LO is conveniently described by means of classical sequents of the form $\Gamma \vdash \Theta$, where Γ is a set of clauses called the *program* and Θ is a multiset of goal formulas called the *context*. The dynamic part of an LO sequent is its context, that will in general contain more than one goal. Once all the goals have been decomposed, a clause is fetched from the program and used to continue the computation. In the following, we will deal with goal formulas containing multiplicative disjunctions (written `@` in LO) of atoms only.

Having multiplicative disjunction built-in, LO is adequate for coding the disjunctive representation of P/T$^\omega$ nets. We represent places as atomic formulas, use multiplicative disjunction to encode markings and a clause of the form `b1 @ ... @ bm :- a1 @ ... @ an` for each transitions $\{\!\{a_1, \ldots, a_n\}\!\} \triangleright \{\!\{b_1, \ldots, b_m\}\!\}$. Since $\bot$ is not available in LO, we simulate it by means of the atomic formula `bot`, making it play the role of the empty multiset whenever a rule has an empty postset. Since LO does not possess a bound context, we encode the initial marking by means of a rule with no postset, distinguished by the atom `stop`.

Figure 4 shows the resulting program for the producer/consumer example.

4.3 Implementation in Forum

Forum [7] was designed as an extension to both Lolli and LO. The fragment of linear logic it mechanizes is the language freely generated from the linear logic symbols $\top$, $\&$, $\bot$, $\parr$, $\multimap$, $\Rightarrow$ and $\forall$. This set of connectives is indeed complete in the sense that the remaining linear symbols can be defined on top of it. The operational semantics of Forum is modelled by means of sequents of the form $\Gamma; \Delta \vdash \Theta$ where the functions of the contexts Γ, Δ and Θ coincide with those of the homonynous entities of Lolli and LO.

```
                                                        Specific rules
r          :- p.                          % t_p
p @ b @ c  :- r.                          % t_r
c          :- b @ b @ a.                  % t_a
a          :- c.                          % t_c
LINEAR r @ n @ n @ b @ b @ b @ a.
                                                        Example query
?- p @ n @ n @ n @ b @ c.
```

Figure 5: A Forum encoding of the producer/consumer net

Figure 5 shows the Forum implementation of the producer/consumer example. In this case, we can adopt in full the disjunctive encoding. The resulting program resembles the LO implementation. Since Forum admits $\bot$ as a primitive connective, there is no need for any trick to simulate the behavior of the empty multiset. Moreover, since there is a distinction between a bound and an unbound part of the program, a termination predicate like `stop` is not required in this implementation. The initial marking is loaded in the bound context by means of the directive `LINEAR`.

5 Conclusions

In this paper, we reported on an experiment with the major linear logic programming languages designed in the recent years: Lolli [5], LO [1] and Forum [7]. As a case study, we considered the representation of a class of Petri nets, P/T$^\omega$ nets, in each of these languages. We observed that the obtained code becomes more direct and elegant as we pass from Lolli to LO, and from LO to Forum. This fact is easily explained by noticing that the set of linear connectives provided by the languages in this list becomes more and more suited to our problem as we proceed.

References

[1] J.M. Andreoli, R. Pareschi: "Linear Objects: Logical Processes with Built-in Inheritance", in *International Conference on Logic Programming* pp. 495–510, 1990.

[2] A. Asperti: "A logic for Concurrency", Manuscript, November 1987.

[3] I. Cervesato: "Petri Nets as Multiset Rewriting Systems in a Linear Framework". Available as http://www.cs.cmu.edu/∼iliano/ON-GOING/deMich.ps.gz.

[4] J.Y. Girard: "Linear Logic", in *Theoretical Computer Science* 50:1–102, 1987.

[5] J. Hodas, D. Miller: "Logic Programming in a Fragment of Intuitionistic Linear Logic", in *Journal of Information and Computation* 110(2):327–365, 1994.

[6] N. Martí-Oliet, J. Meseguer: "From Petri Nets to Linear Logic", in *Conf. on Category Theory and Computer Science*, pp. 313–337, LNCS 389, Springer-Verlag, 1989.

[7] D. Miller: "A Multiple-Conclusion Meta-Logic", in *Symposium on Logic in Computer Science*, pp. 272–281, 1994.

[8] W. Reisig: "*Petri Nets, an Introduction*", Springer-Verlag, 1985.